Chapter 5

Statements

This chapter describes the statements in the VAX C programming language. Statements are executed in the sequence in which they appear in a program, except as indicated. The VAX C statements are grouped as follows:

- Control flow statements
- Expressions and blocks as statements
- Conditional statements
- Looping statements
- Interrupting statements

5.1 Control Flow Statements

You can use some VAX C statements either to maintain or modify the control of the program. The following sections describe the control flow statements.

5.1.1 The null Statement

Use null statements to provide null operations in situations where the grammar of the language requires a statement, but the program requires no work to be done.

The syntax of the null statement is as follows:

```
;
```

You may need to use the null statement with the **if**, **while**, **do**, and **for** statements in cases where the grammar requires a statement body but the program requires no functional operation. The most common use of this statement is in loop operations, where all the loop activity is performed by the test portion of the loop. For example, the following statement finds the first element of an array known to have a value of 0:

```
for(i=0; array[i] != 0; i++)
   ;
```

See Section 5.2 and 5.4 for more information about the statements mentioned here.

5.1.2 The goto Statement

The **goto** statement transfers control unconditionally to a labeled statement, where the label identifier must be located in the scope of the function containing the **goto** statement.

The syntax of the **goto** statement is as follows:

goto identifier;

Take care when branching into a block or function body using the **goto** statement. The compiler allocates storage for automatic variables declared within a block when the block is activated. When a **goto** statement branches into a block, automatic variables declared in the block may not exist in storage. Attempts to access such variables may cause a run-time error. See Chapter 8 for more information about automatic variables.

5.1.3 The label Statement

Labels are identifiers used to flag a location in a program, and to be the target of a **goto** statement.

The syntax of a label is as follows:

identifier:

Any statement can be preceded by a label. The scope of a label is the current function body. Since the label name is independent of the scope rules applied to variables, there can be variables with the same name as the label in the function containing the label. Labels are used only as the targets of **goto** statements.

5.2 Expressions and Blocks as Statements

The statements in the following sections are expressions or groups of statements that you can use when the grammar calls for a single statement.

5.2.1 The expression Statement

You can use any valid expression as a statement by terminating it with a semicolon (;). The following example shows an expression used as a statement:

```
i++;
```

This statement increments the value of the variable i. Note that i++ is a valid VAX C expression that can appear in more complex VAX C statements. See Chapter 6 for more information about the valid VAX C expressions.

5.2.2 The compound Statement

A compound statement in VAX C is often called a block (the compound statement following the parameter declarations in a function definition is called the function body). It allows more than one statement to appear where a single statement is required by the language. The following example shows a block:

```
{
   int x = 5;

   z = 1;
   if (y < x)
      funct(y, z);
   else
      funct(x, z);
}
```

The block contains optional declarations followed by a list of statements, all enclosed in braces. If you include declarations, the variables they declare are local to the block, and, for the rest of the block, they supersede any previous declaration of variables of the same name. Inside blocks, you can initialize variables whose declarations include the **auto**, **register**, **static**, or **globaldef** storage class specifiers.

A block is entered normally when control flows into it, or when a **goto** statement transfers control to a label on the block itself. The compiler-generated code allocates storage for the **auto** or **register** variables each time the block is entered normally; the storage allocations do not occur if a **goto** statement refers to a label inside the block or if the block is the body of a **switch** statement. For more information about storage classes, see Chapter 8.

All function definitions are compound statements.

5.3 Conditional Statements

The statements in the following sections execute only if a tested condition is true.

5.3.1 The if Statement

An **if** statement executes a statement depending on the evaluation of an expression, and may or may not be written with an **else** clause. The syntax of the **if** statement is as follows:

```
if ( expression )
    statement
else
    statement
```

An example of the **if** statement is as follows:

```
if (i < 1)
   funct(i);
else
   {
      i = x++;
      funct(i);
   }
```

If the evaluated expression within parentheses is true (in the example, if variable i is less than 1), then the statement following the evaluated expression executes; the statement following the **else** keyword does not execute. If the evaluated expression is false, then the statement following the **else** keyword executes.

All logical operators define a true result to be nonzero. Therefore, the expression in any conditional statement may be a logical expression with predictable results (true or false; nonzero or zero).

When **if** statements are nested within **else** clauses, each **else** clause matches the most recent **if** statement that does not have an **else** clause.

5.3.2 The switch Statement

The **switch** statement executes one or more of a series of cases, based on the value of the expression.

The syntax of the **switch** statement is as follows:

```
switch ( expression )
    statement
```

The result of the evaluating expression must be of data type **int**. (For more information about the data types, refer to Chapter 7.) The statement is typically a compound statement, where one or more **case** labels prefix statements that execute if the expression matches the **case**. The syntax for a **case** label and expression is as follows:

```
case constant-expression :
    statement[,statement, . . . ]
```

The constant-expression must be of type **int**. No two **case** labels can specify the same value. The value of a constant-expression can be any integral value.

At most one statement in the compound statement can have the following label:

```
default :
```

The **case** and **default** labels can occur in any order. When the **switch** statement is executed, the following sequence takes place (note that each case flows into the next unless explicit action is taken, such as a **break** statement):

1. The **switch** expression is evaluated and compared with the constant expressions in the **case** labels.
2. If the expression matches a **case** label, the statement or list of statements following that label is executed. If the list of statements ends with the **break** statement, the **break** terminates the **switch** statement; otherwise, the next case that is encountered is executed. (See Example 5–1.) You can terminate the **switch** statement by a **return** or **goto** statement; if the **switch** is inside a loop, you can terminate it with a **continue** statement. For more information about interrupting statements, see Section 5.5.
3. If the expression's value does not match any **case** label but there is a **default** case, the **default** case is executed. It need not be the last case listed. If a **break** statement does not end the **default** case and it is not the last case, the next case encountered is executed.
4. If the expression's value does not match any **case** label and there is no **default**, the body of the **switch** statement is not executed.

In general, you must use the **break** statement to ensure that a **switch** statement executes as expected. Example 5–1 uses the **switch** statement to count blanks, tabs, and newlines entered from the terminal.

Example 5–1: Using the switch Statement to Count Blanks, Tabs, and Newlines

```
/*  This program counts blanks, tabs, and newlines in text *
 *  entered from the keyboard.                             */

#include <stdio.h>
main()
{
   int number_tabs = 0, number_lines = 0, number_blanks = 0;
   int ch;
   while ((ch = getchar()) != EOF)
      switch (ch)
         {
❶           case '\t':  ++number_tabs;
❷                       break;
            case '\n':  ++number_lines;
                        break;
            case ' ' :  ++number_blanks;
                        break;
         }
   printf("Blanks\tTabs\tNewlines\n");
   printf("%6d\t%6d\t%6d\n", number_blanks,
                             number_tabs,number_lines);
}
```

Key to Example 5–1:

❶ A series of **case** labels is used to increment the counters.

❷ The **break** statement causes control to go back to the **while** loop every time a counter increments. The program automatically passes control to the **while** loop if none of the counters is incremented.

Example 5–1 responds to the following input:

```
% example RETURN
Every good boy. RETURN
The quick brown fox. RETURN
Line with 2 TAB TAB tabs. RETURN
^D
```

The output is as follows:

```
Blanks      Tabs  Newlines
     7         2         3
```

If you omit the **break** statements, the output is as follows:

```
Blanks      Tabs  Newlines
    12         2         5
```

Without the **break** statements, each case drops through to the next case. The number shown for tabs happens to be right, because the tabs case is first in the **switch** statement and is executed only if the variable ch == '\t'. Notice that the number shown for newlines is the correct number plus the number of tabs, and the number shown for blanks is the total of all three cases.

If variable declarations appear in the compound statement within a **switch** statement, any initializations of the **auto** or **register** variables are ineffective. However, if variables are initialized within the statements following a **case** label, the initialization is effective. Consider the following example:

```
switch (ch)
   {
      int x = 1;              /* Improper initialization */
      printf("%d", x);
      case 'a' :
       { int x = 5;           /*  Proper initialization  */
         printf("%d", x);
         break; }
      case 'b' :
         .
         .
         .
   }
```

In the previous example, if the variable ch equals a, then the program prints the value 5. If the variable equals any other letter, the program prints nothing because the initialization outside of the case label is not executed.

5.4 Looping Statements

The statements in the following sections execute repeatedly (loop), until an expression evaluates to false. Some loops execute a block of statements, known as the loop body, a specified number of times. In VAX C, this loop is the **for** statement. Some loops evaluate an expression and then execute the body of the loop. In VAX C, this loop is the **while** statement. Some loops execute the loop body and then evaluate the expression, which guarantees at least one execution of the body. In VAX C, this loop is the **do** statement.

5.4.1 The for Statement

The **for** statement evaluates three expressions and executes a statement (the loop body) until the second expression evaluates to false. The **for** statement is particularly useful for executing a loop body a specified number of times.

The syntax for the **for** statement is as follows:

```
for ( expression-1 ; expression-2 ; expression-3 )
    statement;
```

The **for** statement executes the loop body zero or more times. It uses three control expressions, as shown. The semicolons (;) separate the expressions. Note that a semicolon does not follow the last expression. A **for** statement performs the following evaluations:

- Expression-1 is evaluated once before the first iteration of the loop. It usually specifies the initial values for variables.
- Expression-2 is a relational or logical expression that determines whether or not to terminate the loop. Expression-2 is evaluated before each iteration. If the expression evaluates to false, execution of the **for** loop body terminates. If the expression evaluates to true, the body of the loop is executed.
- Expression-3 is evaluated after each iteration. It usually specifies increments for the variables initialized by expression-1.
- Iterations of the **for** statement continue until expression-2 produces a false (0) value, or until some statement such as **break** or **goto** interrupts.

The **for** statement is identical to the following syntax:

```
expression-1;
while ( expression-2 )
    {
      statement
      expression-3;
    }
```

The VAX C compiler optimizes certain **for** statements for simple loops. Consider the following example:

```
for(i=0; i<15; i++)
     printf("%d\n", i);
```

When the incrementation is as simple as in the previous example, the compiler generates less macro code so efficiency increases. When possible, use **for** statements instead of **while** statements when the increment is small.

You can omit any of the three expressions in a loop. If you omit expression-2, the test condition is true; that is, the **while** in the expansion becomes **while**(x), where x is not equal to 0. If you omit either expression-1 or expression-3 from the **for** statement, that expression is effectively dropped from the expansion.

The following syntax shows an infinite loop:

```
for (;;) statement
```

Terminate infinite loops with a **break**, **return**, or **goto** statement.

5.4.2 The while Statement

The **while** statement evaluates an expression and executes a statement (the loop body) zero or more times, until the expression evaluates to false.

The syntax of a **while** statement is as follows:

```
while ( expression )
    statement
```

An example of the **while** loop is as follows:

```
while (x < 10)
   {
      array[x] = x;
      x++;
   }
```

The previous example tests the value of the variable x. If variable x is less than 10, it assigns x to the *x*th element of the array and then increments the variable x. If the expression in parentheses evaluates to false, the loop body never executes.

5.4.3 The do Statement

The **do** statement executes a statement (the loop body) one or more times, until the expression in the **while** clause evaluates to false.

The syntax for the **do** statement is as follows:

```
do
    statement
while ( expression ) ;
```

The statement is executed at least once, and the expression is evaluated after each subsequent execution of the loop body. If the expression is true, the statement is executed again.

5.5 Interrupting Statements

You can use the statements in the following sections to interrupt the execution of another statement. These statements are primarily used to interrupt **switch** statements and loops.

5.5.1 The break Statement

The **break** statement terminates the immediately enclosing **while**, **do**, **for**, or **switch** statement. Control passes to the statement following the loop body.

The syntax for the **break** statement is as follows:

```
break;
```

5.5.2 The continue Statement

The **continue** statement passes control to the end of the immediately enclosing **while**, **do**, or **for** statement.

The syntax for the **continue** statement is as follows:

```
continue;
```

Review the following syntax summary to see the effects of the **continue** statement on the looping statements:

```
goto label;
```

The **continue** statement is identical to the **goto** label statement for each of the looping statements in the following syntax examples:

```
while( . . . )
 {
   .
   .
   .
  goto label;
   .
   .
   .
  label:
   ;
 }
```

```
   do
  {
    .
    .
    .
   goto label;
    .
    .
    .
   label:
    ;
  }
  while( . . . )
```

```
for( . . . ; . . . ; . . . )
    {
      .
      .
      .
     goto label;
      .
      .
      .
     label:
      ;
    }
```

In the preceding syntax examples, a **continue** statement passes control to label. The **continue** statement is intended only for loops, not for **switch** statements. A **continue** inside a **switch** statement that is inside a loop causes continued execution of the enclosing loop after exiting from the body of the **switch** statement.

5.5.3 The return Statement

The **return** statement causes a return from a function, with or without a return value.

The syntax of the **return** statement is as follows:

```
return [expression];
```

The compiler evaluates the expression (if you specify one) and returns the value to the calling function. If necessary, the compiler converts the value to the declared type of the calling function's return value. If there is no specified return value, the value is undefined.

You can declare a function without a **return** value to be of type **void**. For more information about the **void** data type and function return values, see Chapter 4.